现在可以编译了！

不可编译协议的经验证自动修复

LuísCruz-Filipe ！
南丹麦大学数学与计算机科学系
Fabrizio Montesi
南丹麦大学数学与计算机科学系

摘要

编排式编程是一种范式，开发者编写通信系统的全局规范（称为编排），随后自动编译出一个正确构造的分布式实现。遗憾的是，有些编排无法编译，这是因为存在与一种称为选择知识的协议一致性属性相关的问题。这迫使程序员手动推理与其所编写协议可能无关的实现细节。

修订是一种自动修复无法编译的编排的过程。我们介绍了文献中的修订形式化，它建立在现有的编排式编程形式化之上。然而，在形式化这一过程预期属性的过程中，我们发现了一个微妙的反例，它推翻了原先发表并经同行评审的纸笔理论。我们讨论了定理证明器如何帮助我们发现这一问题，并陈述和证明修订性质的正确表述。

2012 ACM 学科分类计算理论 → 并发；计算理论→ 自动推理；软件及其工程 → 并发编程语言

关键词和短语 编排式编程、定理证明、编译、程序修复

数字对象标识符 10.4230/LIPIcs...

arXiv:2302.14622v1 [cs.LO] 28 Feb 2023

1 引言

为通信系统编写协议的正确实现颇具挑战，因为这要求为每个参与者编写正确的程序，使其在恰当的时机执行正确的发送和接收动作 [21]。**编排式编程** [24] 是一种新兴范式，提供了直接的解决方案：用一种“编舞式”编程语言编写协议，然后通过一种称为**端点投影**（简称 EPP 或投影）的操作，自动编译出正确的实现 [4, 12, 14, 15, 18, 22, 23]。

编舞语言受到安全协议 [26], 的 Alice 和 Bob 表示法的启发，因为它们提供了用于表达不同进程之间通信的原语。实现通常以进程演算建模。编排式编程不仅简单，而且颇具价值，因为它通常具有强有力的理论保证，最值得一提的是无死锁性，以及编排与生成的分布式实现（模型）之间的操作对应关系。

由于一个称为“选择知识”的问题，并非所有编排都能编译（或投影）为分布式实现 [5]。请看以下简单购买场景的编排（此示例也预先展示了我们的一些语法）。

```
buyer。offer ⟶ seller。x; If seller。acceptable (x)
```

清单 1 一个不可投影的编排。

©Luís Cruz-Filipe 和 Fabrizio Montesi；依据知识共享署名 4.0 国际许可协议授权 莱布尼茨国际信息学会议论文集 达格施图尔城堡—莱布尼茨信息学中心，达格施图尔出版社，德国
LIPICS

XX:2 现在可以编译了!

该编排描述如下： buyer 将其购买产品的报价发送给 seller，后者将报价存储在其局部变量 x中；随后， seller 检查报价是否可以接受，如果可以，则将 product 发送给 buyer。由于 buyer 在条件语句的两个分支中必须表现不同，因此无法将该编排投影为行为等价实现。然而，该条件语句由 seller求值，而 buyer 无法判断最终选择了哪个分支。

编排通常通过添加选择来变得可投影，也就是通信传递称为选择标签的常量。[1] 清单 1 的一个可投影版本如下所示。

```
buyer。offer ⟶ seller。x; If seller。acceptable (x) Then seller ⟶ buyer[left];
```

清单 2 一个可投影的编排。

这个编排与前一个编排的不同之处在于，条件语句的每个分支中都包含一个选择。具体来说，如果 seller 选择了 Then 分支，他们现在就会将标签 left 发送给 buyer。反之，如果选择了 Else 分支，则会改为传递标签 right 。关键在于，现在为 buyer 生成的实现可以读取 seller 收到的标签，从而知道应该执行条件语句的哪个分支。由于标签是常量，编译时可以静态验证 buyer 收到的不同分支标签各不相同，因此具有“选择知识”。

投影可以灵活处理选择知识，将选择的数量降至最低 [2]。只有当某个进程的行为取决于条件语句所选的分支时，它才需要知道选中了哪个分支；如果该进程在条件语句的两个分支中都执行相同的操作，那么这种知识对它就无关紧要。选择知识也可以传递：如果某个进程 q 知道另一个进程 p所做的选择，那么任一进程都可以将这一信息转发给任何需要它的其他进程。

修订。 以往的研究探讨了如何将不可投影的编排自动转换为可投影的编排。这种转换称为修订 [8, 20] 或修复 [1, 13]。例如，对清单 1 中的编排应用 [8] 中的修正过程，会得到清单 2 中的编排（记法上的细微差异除外）。

修订至少有两个方面值得关注。在实践层面，它可以为开发者提供有效的选择策略建议，使其编排能够执行；甚至可以自动完成这一过程，让他们不必担心选择知识。在理论层面，它使得将所有编排集合的完备性性质移植到可投影编排集合成为可能。

后者的一个例子出现在对核心编排（CC）的研究中。CC 是一种极简的编排式编程理论 [8]，我们分两步证明了 CC 中可投影编排集合是图灵完备的。首先，我们证明 CC 是图灵完备的，暂不考虑可投影性问题（证明中构造的编排显然不可投影）；然后，我们定义一个修正过程，并证明编排与其修订之间存在操作对应。因此，可投影编排的子集也具有图灵完备性。类似地，利用可投影编排与其实现之间的操作对应结果，可以证明所使用的进程演算（有状态进程，即 SP）是图灵完备的。

[1] 选择本质上是编排中对应于会话类型里的分支选择，或线性逻辑中的加法联结词的概念 [3, 16]。

问题。我们的初始目标是基于先前对 CC 及其相应投影概念的形式化工作 [8]，在 Coq 定理证明器中对修订及其性质进行形式化 [10] [9]。该形式化采用了基于文献 [25],中理论的 CC 变体，我们发现这种变体更便于形式化。不幸的是，在形式化修订的定义后，我们尝试证明其操作对应结果，但未能成功。检查失败证明的状态后，我们很快发现了一个反例。

原始陈述的不正确性危及依赖于它的后续发展，尤其是可投影编排集合和 SP 的图灵完备性。这些结果对于证实 CC 是编排式编程的一个“良好”极简模型这一主张至关重要。这一发现促使我们提出了一个更具雄心的目标：重新表述关于修订的操作对应，使其正确，同时仍足够强大，能够得到上述结论。

贡献。 据我们所知，这是首次对编舞修订进行形式化。我们在 Coq 定理证明器中陈述并证明了编排及其修订之间操作对应的一个宽松版本，从而增强了对其正确性的信心。我们讨论了使用交互式定理证明器的过程如何帮助我们发现反例，而这些反例引导我们提出了这一考虑所有边界情况的全新且正确的表述。随后，我们利用这一结果，对 [8]，中关于可投影编排和 SP 的图灵完备性证明进行形式化；这些证明未包含在 [10] 中。

论文结构。我们将在第 2 节介绍与核心编舞及其形式化相关的背景知识。第 3 节给出 amend 的定义及其形式化，并讨论和修正 [8]中的操作对应结果。第 4 节表明，修订后的语义性质仍足以推出该工作中的图灵完备性结果。我们将在第 5 节讨论相关工作，并在第 6 节总结全文。

我们的论述假定读者对交互式定理证明有一定了解。文中包含一些 Coq 代码，但本工作旨在使非 Coq 专家也能够理解。

2 背景

我们概述核心编舞的 Coq 形式化的最新版本 [11]。为简化起见，我们省略了对本工作无关紧要的两个要素：该语言以签名为参数，以及通信带有注释（这些注释旨在包含与未来在实际编程语言中实现相关的信息）。这使我们能够省略一些在 amend 的开发中不起作用的子项。

在本文的介绍中，我们使用经过简化的 Coq 表示法，以增强可读性：编排和进程项采用点号重载书写（Coq 表示法机制不允许这样做），而归纳定义和推理规则则使用通常的数学表示法给出。

2.1 核心编排

我们首先概述核心编排（核心编舞），以及在 Coq [10] 中对其进行的形式化。

语法。 CC 的语法由以下语法给出。

XX:4 现在可以编译了！

Type	变量	描述
Choreography	C	编排
Pid	p, q, r, s	进程名称（标识符）
list Pid	ps	进程名称列表
Var	x, y, z	变量名称
Val	v	值
Expr	e	表达式（求值后得到值）
BExpr	b	布尔表达式（求值后得到布尔值）
Label	l	标签（left 和 right）
RecVar	X	过程名称（或递归变量）
DefSet	D	CC 中的过程定义集合
State	s	从变量到值的映射
Configuration	c	带有状态的编舞程序
TransitionLabel	t	转换标签
list TransitionLabel	tl	迁移标签列表
Behaviour	B	行为
option Behaviour	mB	option 行为单子
Network	N	网络
DefSetB	D	SP 中的过程定义集合
Program	P	包含过程定义的编排/网络

表1 原始 Coq 形式化中的类型概要 [10, 9]。

```
C ::= η; C | If p.b Then C1 Else C2 | Call X | RT_Call X ps C | End
η ::= p.e ⟶ q.x | p ⟶ q[l]
```

编排 C 可以是：通信 η 后接一个延续（η; C）；条件语句 If p.b Then C1 Else C2，其中进程 p 计算布尔表达式 b，以便在分支 C1 和 C2之间进行选择；过程调用 Call X，其中 X 是被调用过程的名称；运行时项 RT_Call X ps C;[2]；或已终止的编排 End。通信 η 可以是：值通信 p.e ⟶ q.x，读作“进程 p 在本地计算表达式 e，并将结果发送给进程 q，后者将其存储在本地的 in x 中”；或者选择 p ⟶ q[l]，其中标签 l 可以是 left 或 right，读作“p 向 q发送标签 l ”。

编排在 Coq 中被形式化为一种名为 Choreography 的归纳类型。表1总结了本文使用的 Coq 类型，以及我们对这些类型的元素进行量化时采用的约定。

执行编排需要知道与可调用过程相关联的编排定义，以及这些过程所涉及的进程。过程定义集合定义为从过程名称到由进程名称和编排组成的二元组的映射。

```
Definition DefSet := RecVar →(list Pid)∗Choreography。
```

因此，编舞程序 是由一组过程定义和一个编排组成的有序对（该编排表示“主”编排或“运行中”的编排）。

[2] 出于技术原因，在编排语义的定义中需要运行时项 [10]。这些方面与当前的开发无关。

```
Definition Program:= DefSet * Choreography。
```

我们用 Procedures P 和 Main P 分别表示 P的两个组成部分。程序 P 使用的所有进程的集合定义为 CCP_pn P。

通常会假设编排满足某些良构性条件，例如，任何进程都不会与自身通信。编舞程序还具有额外的良构性条件，这些条件必须对运行时可到达的所有过程都成立。一般来说，这一概念不可判定；但在实际情况下，如果程序只使用有限个过程，则该问题可判定。我们将在第3.2节末尾重新讨论这一点，届时它将变得相关。

▶ **示例1.** 清单1和2中的编排都是良构的。

语义。 核心编舞中的直观系统假设是：进程彼此独立地运行（并发地运行），并拥有局部存储（将其变量与值关联起来）；通信是同步的；网络是可靠的（消息不会丢失或重复，并且在任意两个进程之间都按正确顺序传递）。这些假设源自进程演算，在进程演算中它们相当常见。

▶ **示例2.** 由于进程并发运行，因此可以表达具有并发行为的编排。请看 [25] 中工厂示例的以下简化版本。

```
o.order ⟶ p.x; o'.order' ⟶ p'.y; End
```

清单3 并行订单。

在清单3中，两个进程 o 和 o' 分别向两个不同的供应商 p和 p'下单。由于所有进程彼此不同，且这两次通信之间不存在因果依赖，因此原则上可以按任意顺序执行这两次通信。这为编排带来了乱序执行的概念。

[10] 中编排的语义以配置上的标记迁移系统给出，配置由程序和一个（内存）状态组成。状态为每个进程关联一个从变量名称到值的映射，该映射定义了该进程的内存。

```
Definition State := Pid → Var → Value
```

状态使用一些记号： s [==] s' 表示 s 和 s' 外延相等，而s⟦p,x ⇒ v⟧表示通过映射 p,x ↦ v 更新 s后得到的状态。

有了这些概念，我们就可以展示图1中编舞配置的一些代表性迁移规则。[3] 转移的形式为(D,C,s) —[t]⟶ (D,C',s')，其中 t 是一个转换标签，用于观察迁移中发生了什么。

规则 CC_Com 描述进程 p 向进程 q执行值通信的过程：如果位置 p 处的表达式 e 能够求值为一个值 v （第一个条件，使用辅助函数 eval），那么通信项就会被消耗，接收方的状态也会更新，使其接收变量 x 现在映射到值 v。转换标签 TL_Com p v q 表示 p 已将值 v 传给 q，模拟网络上可见的情况。

[3] 在实际的形式化中，出于技术原因，转换关系分两层定义。这一技术细节与我们的论述无关，因为我们的结果可由此处展示的规则推出。

XX:6 现在可以编译了！

$$\frac{\texttt{v := eval e s p} \quad \texttt{s' [==] s⟦q,x ⇒ v⟧}}{\texttt{(D,p.e ⟶ q.x; C,s)} \xrightarrow{\texttt{[TL_Com p v q]}} \texttt{(D,C,s')}}\ \texttt{CC_Com}$$

$$\frac{\texttt{s [==] s'}}{\texttt{(D,p ⟶ q[l]; C,s)} \xrightarrow{\texttt{[TL_Sel p q l]}} \texttt{(D,C,s')}}\ \texttt{CC_Sel}$$

$$\frac{\texttt{beval b s p = true} \quad \texttt{s [==] s'}}{\texttt{(D,If p.b Then C1 Else C2,s)} \xrightarrow{\texttt{[TL_Tau p]}} \texttt{(D,C1,s')}}\ \texttt{CC_Then}$$

$$\frac{\texttt{beval b s p = false} \quad \texttt{s [==] s'}}{\texttt{(D,If p.b Then C1 Else C2,s)} \xrightarrow{\texttt{[TL_Tau p]}} \texttt{(D,C2,s')}}\ \texttt{C_Else}$$

$$\frac{\texttt{disjoint_eta_rl } \eta\ \texttt{t} \quad \texttt{(D,C,s)} \xrightarrow{\texttt{[t]}} \texttt{(D,C',s')}}{\texttt{(D,}\eta\texttt{; C,s)} \xrightarrow{\texttt{[t]}} \texttt{(D,}\eta\texttt{; C',s')}}\ \texttt{CC_Delay_Eta}$$

$$\frac{\texttt{disjoint_p_rl p t (D,C1,s)} \xrightarrow{\texttt{[t]}} \texttt{(D,C1',s')} \quad \texttt{(D,C2,s)} \xrightarrow{\texttt{[t]}} \texttt{(D,C2',s')}}{\texttt{(D,If p.b Then C1 Else C2,s)} \xrightarrow{\texttt{[t]}} \texttt{(D,If p.b Then C1' Else C2',s')}}\ \texttt{CC_Delay_Con}$$

d

图1 编舞配置的语义（选定规则）。

规则 CC_Sel 类似，但不会改变接收方的状态（等我们解释编舞实现建模语言时，选择的作用会更加清楚）。转换标签 TL_Sel p q l 记录了从 p 到 q的标签 l 通信。

规则 CC_Then 处理进程 p 能够求值条件语句守卫条件 b 并得到 true 的情况（使用辅助函数 beval），从而继续执行条件语句的 then 分支。转换标签 TL_Tau p 表示进程 p 执行了一个内部动作（τ 是进程演算中此类动作的标准符号）。

规则 CC_Delay_Eta 处理通信的乱序执行，对示例2中预先提出的推理进行形式化。具体而言，只要迁移不涉及 η中的任何进程，交互 η 的延续就可以执行迁移（且不影响 η）。规则的第一个前提disjoint_eta_rl η t检查 η 中提及的进程是否不同于转换标签 t所提及的进程，以此验证上述条件。规则 CC_Delay_Cond 将相同的推理应用于条件语句的乱序执行。

转换关系的自反传递闭包记作 $\xrightarrow{\texttt{[tl]}}{}^*$，其中tl 是转换标签列表。

▶ **示例3.** 对于任意 D 和 s，若 order 在 o 处求值为 v，且 order' 求值为 o v' 在 o'处，根据 eval，

$$\texttt{(D,o.order} \longrightarrow \texttt{p.x; o'.order'} \longrightarrow \texttt{p'.y; End,s)} \xrightarrow{\texttt{[TL_Com o v p;TL_Com o' v' p']}}{}^* \texttt{(D,End,s')}$$

and

$$\texttt{(D,o.order} \longrightarrow \texttt{p.x; o'.order'} \longrightarrow \texttt{p'.y; End,s)} \xrightarrow{\texttt{[TL_Com o' v' p';TL_Com o v s]}}{}^* \texttt{(D,End,s')}$$

其中 s' [==] s⟦s1,x ⇒ v⟧⟦s2,x ⇒ v'⟧。

2.2 进程

编排的实现采用有状态进程（SP） [9], 建模。SP 是一种形式化的进程演算，遵循 [25]。SP 遵循表示通信进程系统的标准方式，

即分别给出每个进程的代码，并在进程执行兼容的输入/输出操作时实现通信。

语法。 进程的代码写作行为（B），遵循以下语法。

```
B ::= p!e; B | p?x; B | p+l; B | p & mB1 //
```

这些项是 CC 中编舞项在局部层面的对应形式。前两条产生式处理值通信。具体来说，发送动作 p!e; B 会将对 e 求值的结果发送给进程 p，然后继续执行 B。与之对偶的是，接收动作 p?x; B 会从 p 接收一个值，将其存入 x，然后继续执行 B。

选择通过原语 p+l; B 和 p & mB1// mB2 实现。前者会将标签 l 发送给进程 p，然后继续执行 B。后者是一个分支项，其中 mB1 和 mB2 是进程分别在接收到 left 或 right 后执行的行为。为涵盖进程未针对某个特定标签提供行为的情况，mB1 和 mB2 的类型为 option Behaviour。

条件语句（If b Then B1 Else B2）、过程调用（Call X）和终止行为（End）都是标准构造。

进程旨在网络中协同运行。网络被形式化为从进程到行为的映射。

```
Definition Network := Pid → Behaviour。
```

网络提供了一些便于构造网络的记法：p[B] 表示将 p 映射到 B，并将所有其他进程映射到 End；而 N | N' 表示 N 和 N'的组合。特别是，(N | N') p 在其不同于 End时返回 N p，否则 N' p 返回 [4]。

▶ **例 4.** 以下网络实现了列表 2 中的编排。

```
buyer[ seller!offer; seller & Some (seller?y; End) // Some End ] |
```

关于网络的语义，我们还需要两个额外的要素。网络 N \ p由 N 通过重新定义 p的行为而得到：End (p 从 N中“移除”)。当网络 N 和 N' 外延相等时，关系N (==) N' 成立。

与核心编舞中一样，网络中的进程可以调用在单独集合中定义的过程。

```
Definition DefSetB : = RecVar → Behaviour。
```

SP 中的一个 Program 由一组过程定义和一个网络组成。

```
Definition Program: = DefSetB * Network。
```

我们使用 D 表示 DefSetB 的元素，并使用 P 表示 Program的元素，这与编排中的用法相同（区别将从上下文中明确）。

语义。 与核心编舞中一样，SP 的语义也以标记迁移系统的形式给出，该系统作用于由程序和内存状态组成的配置。图 2 展示了定义这一语义的部分迁移规则。

[4] 这种不对称性不会影响我们的结果，因为我们从未组合过会为相同进程定义未终止行为的网络。

XX:8 现在可以编译了！

$$\frac{\texttt{N p} = \texttt{q!e; B} \quad \texttt{N q} = \texttt{p?x; B'} \qquad \texttt{v} := \texttt{eval e s p} \qquad \texttt{N'} \ (==)\ \texttt{N} \setminus \texttt{p} \setminus \texttt{q} \mid \texttt{p[B]} \mid \texttt{q[B']} \quad \texttt{s'} \ [==]\ \texttt{s}[\![\texttt{q,x} \Rightarrow \texttt{v}]\!]}{(\texttt{D,N,s}) \ \longrightarrow\!\![\texttt{TL_Com p v q}]\!\longrightarrow \ (\texttt{D,N',s'})} \ \texttt{SP_Com}$$

$$\frac{\texttt{N p} = \texttt{q+left; B} \quad \texttt{N q} = \texttt{p \& Some Bl // mBr} \qquad \texttt{N'} \ (==)\ \texttt{N} \setminus \texttt{p} \setminus \texttt{q} \mid \texttt{p[B]} \mid \texttt{q[Bl]} \quad \texttt{s} \ [==]\ \texttt{s'}}{(\texttt{D,N,s}) \ \longrightarrow\!\![\texttt{TL_Sel p q left}]\!\longrightarrow \ (\texttt{D,N',s'})} \ \texttt{SP_LSel}$$

$$\frac{\texttt{N p} = \texttt{q+right; B} \quad \texttt{N q} = \texttt{p \& mBl // Some Br} \qquad \texttt{N'} \ (==)\ \texttt{N} \setminus \texttt{p} \setminus \texttt{q} \mid \texttt{p[B]} \mid \texttt{q[Br]} \quad \texttt{s} \ [==]\ \texttt{s'}}{(\texttt{D,N,s}) \ \longrightarrow\!\![\texttt{TL_Sel p q right}]\!\longrightarrow \ (\texttt{D,N',s'})} \ \texttt{SP_RSel}$$

$$\frac{\texttt{N p} = \texttt{If b Then B1 Else B2} \quad \texttt{beval b s p} = \texttt{true} \qquad \texttt{N'} \ (==)\ \texttt{N} \setminus \texttt{p} \mid \texttt{p[B1]} \quad \texttt{s} \ [==]\ \texttt{s'}}{(\texttt{D,N,s}) \ \longrightarrow\!\![\texttt{TL_Tau p}]\!\longrightarrow \ (\texttt{D,N',s'})} \ \texttt{SP_Then}$$

图 2 网络配置语义（部分规则）。

规则 SP_Com 将进程 p 上的发送动作与另一进程 q 上的兼容接收动作相匹配（条件为 N p = q! e; B 和 N q = p? x; B'）。结果网络 N' 由 N 通过将这些进程的行为替换为其延续（N\ p\ q | p[B] | q[B']）而得到。状态更新的处理方式与核心编舞中相同。

规则 SP_LSel 及其对偶规则 SP_RSel 分别根据发送方发送的标签，决定分支项所提供的左分支和右分支。规则SP_Then 描述条件语句进入 then 分支的情形。

2.3 端点投影（EPP）

编排通过一种称为行为投影的过程编译为网络。该过程是一个偏函数；由于 Coq 中的所有函数都是全函数，因此将其形式化为如下归纳关系。

```
bproj :  DefSet → Choreography → Pid → Behaviour → Prop
```

术语 bproj D C p B，写作 ⟦D,C | p⟧ == B，读作“在过程定义集 D 的上下文中， C 在 p 上的投影是 B”。[5]

直观地说，行为投影通过遍历编排来计算；对于每个编排项，投影都会构造输入进程为实现它而应执行的局部动作。定义 bproj 的规则中，与本文相关的是处理选择和条件语句的规则。这些规则如图 3 所示。

标签选择 p ⟶ q[l] 会被投影为以下之一：(i) 向进程 p 发送标签 l q （规则 bproj_Pick）；(ii) 适当的分支项，从进程 p 接收 l q，其中只有对应于 l 的分支才提供行为（规则 bproj_Left 及其对偶规则 bproj_Right）；或 (iii) 对任何其他进程均不执行操作（规则 bproj_Sel）。

[5] 参数 D 用于投影过程调用，这与我们的开发无关。

$$\frac{\llbracket \mathtt{D}, \mathtt{C} \mid \mathtt{p} \rrbracket == \mathtt{B}}{\llbracket \mathtt{D}, \mathtt{p}.\mathtt{e} \longrightarrow \mathtt{q}.\mathtt{x}; \mathtt{C} \mid \mathtt{p} \rrbracket == \mathtt{q}!\mathtt{e}; \mathtt{B}}\ \mathtt{bproj_Send} \qquad \frac{\mathtt{p} \neq \mathtt{q} \quad \llbracket \mathtt{D}, \mathtt{C} \mid \mathtt{q} \rrbracket == \mathtt{B}}{\llbracket \mathtt{D}, \mathtt{p}.\mathtt{e} \longrightarrow \mathtt{q}.\mathtt{x}; \mathtt{C} \mid \mathtt{q} \rrbracket == \mathtt{p}?\mathtt{x}; \mathtt{B}}\ \mathtt{bproj_Recv}$$

$$\frac{\mathtt{p} \neq \mathtt{r} \quad \mathtt{q} \neq \mathtt{r} \quad \llbracket \mathtt{D}, \mathtt{C} \mid \mathtt{r} \rrbracket == \mathtt{B}}{\llbracket \mathtt{D}, \mathtt{p}.\mathtt{e} \longrightarrow \mathtt{q}.\mathtt{x}; \mathtt{C} \mid \mathtt{r} \rrbracket == \mathtt{B}}\ \mathtt{bproj_Com}$$

$$\frac{\llbracket \mathtt{D}, \mathtt{C} \mid \mathtt{p} \rrbracket == \mathtt{B}}{\llbracket \mathtt{D}, \mathtt{p} \longrightarrow \mathtt{q}[\mathtt{l}]; \mathtt{C} \mid \mathtt{p} \rrbracket == \mathtt{q}+\mathtt{l}; \mathtt{B}}\ \mathtt{bproj_Pick} \qquad \frac{\mathtt{p} \neq \mathtt{r} \quad \mathtt{q} \neq \mathtt{r} \quad \llbracket \mathtt{D}, \mathtt{C} \mid \mathtt{r} \rrbracket == \mathtt{B}}{\llbracket \mathtt{D}, \mathtt{p} \longrightarrow \mathtt{q}[\mathtt{l}]; \mathtt{C} \mid \mathtt{r} \rrbracket == \mathtt{B}}\ \mathtt{bproj_Sel}$$

$$\frac{\mathtt{p} \neq \mathtt{q} \quad \llbracket \mathtt{D}, \mathtt{C} \mid \mathtt{q} \rrbracket == \mathtt{B}}{\llbracket \mathtt{D}, \mathtt{p} \longrightarrow \mathtt{q}[\mathtt{left}]; \mathtt{C} \mid \mathtt{q} \rrbracket == \mathtt{p}\ \&\ \mathtt{Some\ B}\ //\ \mathtt{None}}\ \mathtt{bproj_Left}$$

$$\frac{\mathtt{p} \neq \mathtt{q} \quad \llbracket \mathtt{D}, \mathtt{C} \mid \mathtt{q} \rrbracket == \mathtt{B}}{\llbracket \mathtt{D}, \mathtt{p} \longrightarrow \mathtt{q}[\mathtt{right}]; \mathtt{C} \mid \mathtt{q} \rrbracket == \mathtt{p}\ \&\ \mathtt{None}\ //\ \mathtt{Some\ B}}\ \mathtt{bproj_Right}$$

$$\frac{\llbracket \mathtt{D}, \mathtt{C1} \mid \mathtt{p} \rrbracket == \mathtt{B1} \quad \llbracket \mathtt{D}, \mathtt{C2} \mid \mathtt{p} \rrbracket == \mathtt{B2}}{\llbracket \mathtt{D}, \mathtt{If\ p.b\ Then\ C1\ Else\ C2} \mid \mathtt{p} \rrbracket == \mathtt{If\ b\ Then\ B1\ Else\ B2}}\ \mathtt{bproj_Cond}$$

$$\frac{\mathtt{p} \neq \mathtt{r} \quad \llbracket \mathtt{D}, \mathtt{C1} \mid \mathtt{p} \rrbracket == \mathtt{B1} \quad \llbracket \mathtt{D}, \mathtt{C2} \mid \mathtt{p} \rrbracket == \mathtt{B2} \quad \mathtt{B1}\ [\mathtt{V}]\ \mathtt{B2} == \mathtt{B}}{\llbracket \mathtt{D}, \mathtt{If\ p.b\ Then\ C1\ Else\ C2} \mid \mathtt{p} \rrbracket == \mathtt{B}}\ \mathtt{bproj_Cond'}$$

图 3 行为投影的选定规则。

类似地，编排中的条件语句会被投影为由守卫条件求值的进程所执行的条件语句（规则 `bproj_Cond`）。然而，考虑到其他进程时，条件语句的投影会变得复杂，因为这需要处理第1节中讨论的选择知识问题。此情况由规则 `bproj_Cond`的处理：如果合并有定义，则将投影结果设为两个分支的投影的“合并”，记作 `B1 [V] B2 == B`。

直观地说，合并试图从两个结构相似但在分支项中所接受的标签上可能存在差异的行为 `B1` 和 `B2` 构建一个行为 `B` 。对于除分支以外的所有项，合并要求项相等，然后以同态方式处理子项。图 4 中的规则 `merge_End`、 `merge_Sel` 和`merge_Cond` 展示了这一点。

有趣的部分在于分支项的合并，它针对每一种可能的组合都有一条规则。图 4 展示了两个具有代表性的情况。如果两个分支项包含针对不同标签的分支，那么我们会得到一个分支项，其中两个分支按照规则 `merge_Branching_SNNS` 所示进行合并。如果两个分支项包含重叠的分支，那么我们会尝试按照规则 `merge_Branching_SSSS`所示将它们合并。[6]

正如我们所指出的，合并（视为一个偏函数）可能未定义，例如 `End` 和 `p+l; End` 无法合并。这引出了第1节中预先提出的可投影性 概念：如果对于这些参数， `bproj` 有定义，则编排 `C` 在过程 `p` 上、相对于一组过程定义 `D` 而言是可投影的。

```
Definition projectable_B D C p := ∃ B, ⟦D,C | p⟧ == B。
```

这由 `projectable_C D C ps` 推广而来，它指出，对于列表 `ps` 中的所有进程， `C` 都是可投影的。若要使编舞程序 `P` 可投影，记作 `projectable_P P`，我们

[6] 由于篇幅限制，图 4 中对这些规则的名称进行了缩写。

XX:10 现在可以编译了！

$$\frac{}{\texttt{End}\ [\texttt{V}]\ \texttt{End} == \texttt{End}}\ \text{merge_End} \qquad \frac{\texttt{B1}\ [\texttt{V}]\ \texttt{B2} == \texttt{B}}{\texttt{p+1; B1}\ [\texttt{V}]\ \texttt{p+1; B2} == \texttt{p+1; B}}\ \text{merge_Sel}$$

$$\frac{\texttt{Bt1}\ [\texttt{V}]\ \texttt{Bt2} == \texttt{Bt} \quad \texttt{Be1}\ [\texttt{V}]\ \texttt{Be2} == \texttt{Be}}{\texttt{If p.e Then Bt1 Else Bt2}\ [\texttt{V}]\ \texttt{If p.e Then Be1 Else Be2} == \texttt{If p Then Bt Else Be}}\ \text{merge_Cond}$$

$$\frac{}{\texttt{p \& None // None}\ [\texttt{V}]\ \texttt{p \& None // None} == \texttt{p \& None // None}}\ \text{NNNN}$$

$$\frac{}{\texttt{p \& Some bL // None}\ [\texttt{V}]\ \texttt{p \& None // None} == \texttt{p \& Some bL// None}}\ \text{SNNN}$$

$$\frac{}{\texttt{p \& Some bL // None}\ [\texttt{V}]\ \texttt{p \& None // Some bR} == \texttt{p \& Some bL// Some bR}}\ \text{SNNS}$$

$$\frac{\texttt{bL1}\ [\texttt{V}]\ \texttt{bL2} == \texttt{bL}}{\texttt{p \& Some bL1 // None}\ [\texttt{V}]\ \texttt{p \& Some bL2// None} == \texttt{p \& Some bL // None}}\ \text{SNSN}$$

$$\frac{\texttt{bL1}\ [\texttt{V}]\ \texttt{bL2} == \texttt{bL}}{\texttt{p \& Some bL1 // Some bR}\ [\texttt{V}]\ \texttt{p \& Some bL2// None} == \texttt{p \& Some bL // Some bR}}\ \text{SSSN}$$

$$\frac{\texttt{bL1}\ [\texttt{V}]\ \texttt{bL2} == \texttt{bL} \quad \texttt{bR1}\ [\texttt{V}]\ \texttt{bR2} == \texttt{bR}}{\texttt{p \& Some bL1 // Some bR1}\ [\texttt{V}]\ \texttt{p \& Some bL2 // Some bR2} == \texttt{p \& Some bL // Some bR}}\ \text{SSSS}$$

图 4 合并关系的定义（部分规则）。

要求对于 `CCP_pn P` 中的所有进程， `Main P` 都是可投影的，并且所有过程对于其使用的进程都是可投影的。

有了可投影性之后，端点投影（EPP）被定义为一个函数，它将可投影的编排程序映射为 SP 中的进程程序。

```
Definition epp P : projectable_P P → Program。
```

`epp` 的第二个参数是对 `projectable_P P`的证明，但可以证明结果并不依赖于该项。

▶ **例 5**。 `buyer` 和 `seller` 在例 4 中的行为，分别是列表 2 中编排对于这两个进程的投影。

2.4 图灵完备性

[10] 的作者形式化证明了 CC 是图灵完备的，即所有克莱尼部分递归函数 [19] 都可以针对某种适当的实现概念实现为一种编排。该证明很有趣，因为它考虑了在进程上实例化的、计算能力受到严格限制的 CC：值是自然数；表达式只能是常数零、变量或变量的后继；布尔表达式只能检查某个进程中的两个变量是否包含相同的值。随后，通过让进程按照适当的模式进行通信，将克莱尼部分递归函数并发地实现出来。

编舞程序 `P` 实现 `f: PRFunction m` （表示部分递归函数 $f\colon \mathbb{N}^m \to \mathbb{N}$），其输入进程为 $\texttt{ps}_1,\ldots,\texttt{ps}_m$，输出进程为 `q`，当且仅当：对于任意状态 `s`，其中 $\texttt{ps}_1,\ldots,\texttt{ps}_m$ 的变量 `x` 中分别存储值 $\texttt{n}_1,\ldots,\texttt{n}_m$，(i) 若 $f(\texttt{n}_1,\ldots,\texttt{n}_m) = \texttt{n}$，

则从 s 开始的 P 的所有执行都会终止，并且终止时处于这样一个状态： q 的变量x中存储着 n ； 而 (ii) 若 $f(n_1,\ldots,n_m)$ 未定义，则从 s 开始执行 P 永远不会终止。[7] 这由 Coq 项 implements P m f ps q 表示，其中 ps 是向量 $ps_1,\ldots,ps_m$。

图灵完备性的证明将偏递归函数编码为编排；这些编排并非总是可投影的，因为它们不包含选择，但有些进程在条件语句中的行为各不相同。

3 修订

多项研究探讨了如何自动修订不可投影的编排，以获得可投影的版本 [1, 8, 20]。具体而言，[8] 提出了一种基于合并的修正过程。其思路是，每当编排中包含条件语句时，修订便会在两个分支中都添加选择，从负责计算守卫条件的进程发送到行为投影未定义的任意进程。直观而言，这使输出的编排变得可投影。

▶ **例6**。 令 C 为以下编排：

```
p。 e ⟶ q。 x;  If r。 b Then (r。 e' ⟶ p。 y;  End) Else End
```

按照所述方式对 C 进行修订，可得到以下编排 A：

```
p.e ⟶ q.x; If r.b Then (r ⟶ p[l]; r.e' ⟶ p.y; End)
                   Else (r ⟶ p[r]; End)
```

据称，修订具有以下性质。

▶ **引理 7** （修正引理 [8],改述）。对于每个编排 C：
1. 对 C 的修订是良构的。*2.* 对 C 的修订是可投影的。*3.* 如果 DA、 A和 A'是通过修订 D 中的所有过程以及 C 和 C'得到的，那么对于某个 tl'，$(D, C, s) \xrightarrow{[tl]}{}^{*} (D, C', s')$当且仅当$(DA, A, s) \xrightarrow{[tl']}{}^{*} (DA, A', s')$。

第一点中的良构性指一组句法条件，这些条件可排除编写不当的编排，例如自通信（进程与自身通信的交互） [8]。第一点和第二点很容易通过对编排结构进行归纳法证明。遗憾的是，第三点是错误的。在尝试将这一结果形式化时，我们未能成功，而证明的进展情况使我们得到了以下反例。

▶**例8.** 给定一个合适的状态，来自例6的编排 C 可以迁移到 C' ，定义为

```
p.e ⟶ q.x; r.e' ⟶ p.y; End
```

根据规则 CC_Delay_Eta 和 CC_Then。然而， C的修订 A 可以迁移到

```
p.e ⟶ q.x; r ⟶ p[l]; r.e' ⟶ p.y; End
```

按照相同的规则，但这既不是对 C' 的修订，也无法到达它，因为有问题的选择项被初始通信阻断了。

[7] 这是对图灵机函数实现定义的一种直接改编 [28]。

XX:12 现在可以编译了！

事后看来，这并不那么令人意外：amend 会引入源编舞中原本不存在的因果依赖。然而，最初讨论修订 [7, 8] 的论文作者和审稿人都完全忽略了这一点。因此，修订编舞可能会移除某些执行路径。

在本节余下部分中，我们将展示如何在 Coq 中形式化定义修订，并给出引理 7 的一个正确变体。

3.1 定义

我们将修订的定义分解为三个函数：一个用于识别需要获知特定条件语句结果的进程；一个用于在编排前添加一列选择；另一个则利用前两个函数递归地修订整个编排。这样的划分不仅简化了定义，也简化了有关修订的证明结构，因为证明可以模块化。

为了识别需要具备选择知识的进程，我们定义一个函数 `up_list`（up（不可投影进程的简称）。该函数递归遍历进程列表 `ps`，并逐一检查列表中的每个进程是否能够投影该编排 `If p。 b Then C1 Else C2`（函数 `projectable_B_dec` 正是用于执行此项检查）。如果不能投影，则将该进程加入结果中。（由于可投影性取决于一组过程定义，因此也需要将其作为参数提供， `D`。）

```
Fixpoint up_list D p b ps C1 C2 : list Pid :
```

注意，作为条件语句求值者的 `p` 无需获知结果。这解释了为何要进行检查 `r =? p`；纳入此检查还能避免引入自通信，并简化后续证明。

第二个要素很简单：给定一个进程 `p`、一个选择标签 `l` 和一个编排 `C`，它会递归地将从 `p` 发送 `l` 的选择添加到列表 `ps` 的每个元素中。

```
Fixpoint add_sels p l ps C :  Choreography := match ps with
  | nil ⇒ C
  | r ::  ps' ⇒ p ⟶ r[l]; add_sels p l ps' C end.
```

现在，我们可以按照 [8] 中描述的非正式过程来定义修订。给定一个进程列表 `ps`，我们遍历一个编排 `C`；每当遇到进程 `p` 上的条件语句时，就从 `ps` 中计算出投影未定义的进程列表，并在条件语句的分支前置入适当的选择。（这里只展示最有趣的情况。）

```
Fixpoint amend D ps C : = match C with| eta;  C'  ⇒ eta;  (
```

修订以显而易见的方式推广到过程定义集合。

```
Definition amend_D D ps : DefSet := fun X ⇒(fst (D X), amend D ps (snd (D X)))。
```

要修改程序 P，前述函数的参数 ps 将实例化为 P中使用的进程集合。

```
Definition amend P P :=_
  (amend_D (Procedures P) (CCP_pn P), amend(Procedures P) (CCP_pn P) (Main P)).
```

这个形式化定义对应于 [8] 中给出的非形式化定义。特别是，我们的所有示例都在 Coq 中形式化。

▶ **示例 9.** 考虑以下编排。

```
If p.b Then (p.e ⟶ q.x; q.e' ⟶ r.y; End)
```

这里， p 决定是否 (i) 向 q 发送一个值，该值可用于计算稍后由 q 发送给 r 的消息（因此 q 充当某种代理）；或者 (ii) 由 q 自行计算它将发送给 r 的值。amend 足够智能，能够注意到，虽然 q 需要从 p 接收一个选择，但 r 不需要，因为它的行为方式相同（从 q 在 x 上接收）。因此，修订该编排后会得到以下结果。

```
If p。 b Then （p ⟶ q[left]; p。 e ⟶ q。 x; q。 e' ⟶ r。
```

3.2 句法性质

现在我们讨论修订的关键性质。

修订保持编排（Choreography_WF）和编排程序（Program_WF）的良构性。这是因为 add_sels 通过归纳法保持了良构性的所有句法性质。

```
Lemma amend_Choreography_WF : Choreography_WF C → Choreography_WF （amend D ps C）。

Lemma amend_Program_WF : Program_WF （D, C） → Program_WF （amend_D D ps, amend D ps C）。
```

（为简洁起见，我们省略引理开头的全称量词。）

同样，证明针对某些进程进行 amend 可保证输出的编排在所有这些进程上都是可投影的，是很直接的。

```
Lemma amend_projectable_C : projectable_C （amend_D D ps） （amend D ps C） ps。
```

我们不会将这一结果推广到编舞程序：这样做并不直接，而我们后续的开发也不需要这一结果。我们遇到的问题与 [10, 9] 中讨论的问题有关：计算编排所使用的进程和过程集合可能需要无限多步，因此无法在 Coq 中定义为一个函数。（一个简单的例子是：程序包含一个无限的过程定义集合，其中每个过程 X_i 都会调用下一个过程 X_{i+1}。）

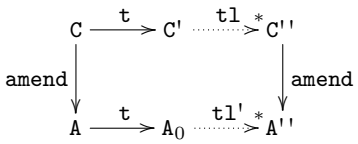
定义 amend_P 时使用的函数 CCP_pn 确实会返回参与程序 P的进程集合，但它不会检查 P 是否定义了未使用的过程。若确实如此，这些过程可能会使用不在 CCP_pn P中的进程，因此对于这些进程而言，它们可能是不可投影的。与其陈述一个带有复杂附加条件的结果

作为假设，我们通过将 amend_projectable_C应用于 Main P 以及所有过程定义的主体，来证明特定程序的可投影性。下一节的发展采用了这一策略。

3.3 语义性质

现在我们讨论如何表述编排与其 s
修订之间的语义关系需要如何改变。

前面展示的反例表明，可以允许两个编排都执行附加转移，以解除阻塞并移除由修订引入的遗留选择。（在我们的示例中，这将是从 p 到 q 的通信。）相应关系将如下所示，其中虚线表示存在量化项：



而转移标签列表 tl 可以通过从 tl' 中移除一些选择来获得。

我们尝试证明这个结果时发现，除了一个情况之外，它在所有情况下都成立：当迁移 t 是通过应用规则 CC_Delay_Cond得到时。

▶ **示例 10.** 我们给出一个最小反例。考虑编排

```
If p.b Then (q.e ⟶ r.x; q.e ⟶ p.x; End)
       Else (q.e ⟶ r.x; End)
```

及其修订

```
If p.b Then (p ⟶ q[left]; q.e ⟶ r.x; q.e ⟶ p.x; End)
       Else (p ⟶ q[right]; q.e ⟶ r.x; End) .
```

原始编排可以执行 q 与 r之间的通信，从而到达

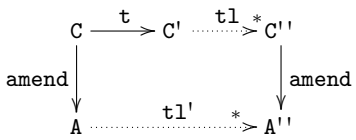
```
If p.e Then (q.e' ⟶ p.x; End) Else End
```

但其修订版必须先执行条件语句和一次选择，才能执行相同的通信。

解决这个问题有两种方法：修改修订的定义，或者进一步细化对应结果。我们选择了第二种方法，原因有二：首先，这样我们就能保留文献 [8]中给出的原始定义；其次，要让修订足够智能，能够识别这类情况，就需要对编排进行非局部分析（即检查条件语句各分支的结构，而不是仅仅检查项是否具有可投影性）。在我们的例子中，这种分析可以发现，只有在 q 到 r的通信之后，才能添加从 p 到 q 的额外选择，从而解决这个问题。

因此，我们最终的对应结果要求，编排的修订版本必须能够在匹配原始编排所执行的迁移之前 执行附加转移。由于一次迁移可能会多次调用规则 c_delay_Cond ，这意味着，忽略额外的选择后，原始编排及其修订版本所执行的转移顺序可以互为任意排列。

我们证明的对应结果如下：



其中 `t::tl` 可以通过移除一些选择并重新排列标签，从 `tl'` 得到。为了在 Coq 中形式化这一点，我们在迁移标签列表之间引入一个关系 `sel_exp` （“选择扩展”）。

```
Inductive sel_exp: ⫤ se_base tl tl' :  Permutation tl tl'
```

现在我们可以证明编排及其修订之间对应关系的正确版本。总共有四个结果：上图所示的结果及其推广，即将 `t` 替换为迁移标签列表的情况；以及两个对偶结果，其中编排的修订先执行。我们证明这两个更一般的命题。

```
Lemma amend_complete_many : Program_WF (D,C) →(D,C,s)  —[tl]⟶* (D,C',s') →∃ tl'  tl


Lemma amend_sound_many :  Program_WF (D,C) → let (D': = amend_D D ps) in(D',
```

本节工作的难点在于理解这些结果的正确表述应该是什么。一旦得到这种表述，证明就相对直接了：对给定的迁移进行归纳（每个情形需要 10–15 行 Coq 代码）。

修订引理的形式化包括 6 个定义、50 个引理和 4 个示例，总计约 1050 行 Coq 代码。

4 修订的含义

在上一节中，对于 [8] 中修订所保证的语义对应关系，我们不得不弱化原始表述。由于原始表述曾用于证明可投影核心编排和 SP 的图灵完备性，因此有必要探究我们的新表述是否仍能得出这些结果。

为保持统一，我们首先重新表述 [10]，中关于核心编排图灵完备性的结果，其中进程名称被视为自然数。

```
Theorem CC_Turing_Complete :  ∀ n (f: PRFunction n),
```

该定理指出，对于任意部分递归函数 `f`，都存在一个良构编舞程序 `P`，它实现 `f`，其输入进程为 1、……、 `n`，输出进程为0。证明可由 [10] 中已给出的结果直接组合得到。

将这一结果与我们关于 amend 的引理结合起来，可得可投影核心编舞的片段同样是图灵完备的。

XX：16 现在可以编译了！

```
Lemma projCC_Turing_Complete： ∀n （f: PRFunction n）， ∃P,
```

证明分为几个步骤。最有趣的子引理证明了：修订一个实现某函数的编排后，所得编排仍实现同一个函数。这被表述为关于修订的一般性结果。

```
Lemma amend_implements  Program_WF P →： （amend_P P） f ps q。
```

证明利用已终止的编排无法继续执行这一事实，说明修正引理添加到原编排中的额外转移列表（tl 见最后一个示意图）必定为空。

关于 projCC_Turing_Complete的其余引理讨论的是修订后编排的可投影性，正如上一节所述，证明起来很简单。

由于修订后的编排是可投影的，我们可以进一步应用 [9] 中的端点投影定理，证明 SP 也是图灵完备的。

```
Theorem SP_Turing_Complete： ∀n (f: PRFunction n)， ∃P,
```

SP_implements 的定义是对编排中 implements定义的直接改编。 SP_Turing_Complete 的证明采用了与 projCC_Turing_Complete相似的策略：我们证明一个子引理 epp_implements ， 指出实现函数 f 的编排的 EPP 是一个实现 f的进程程序。

本节的形式化包含 2 个定义和 11 个引理，总计约 250 行 Coq 代码。这项工作的简洁性印证了我们此前的评论：在第3.2节结尾处，我们没有为程序的可投影性提供复杂的引理。

5 相关工作

据我们所知，我们的工作是首个对修订及其性质和预期后果进行形式化的研究。

与我们工作最接近的是最初提出修正过程的论文，它启发了我们 [8]。正如我们所讨论的，作者所陈述的修正行为对应是错误的。我们提出了正确的表述，并相应地更新和形式化了 CC 和 SP 图灵完备性的证明。我们对行为对应的形式化还阐明了修正实际保证了何种语义性质，这对修正的未来研究和实际应用可能很重要。

其他编舞语言也研究过修正或类似过程 [1, 20]。在所有这些工作中，总体思路都是按需添加通信，使编排变得可投影。然而，底层语言之间的差异使这些过程与我们基于合并的方法大不相同。合并最早在 [2]中提出。

我们的工作基于 CC、SP 和 EPP 形式化的最新版本 [11]，该形式化最初于 [9, 10] 中提出。为了将其用于我们的开发，我们无需修改这一形式化；这表明它已足够成熟，可以作为一个库来推理编排。

其他编排形式化工作包括：Kalas，一种面向 CakeML 的编舞编程语言 [27]；编舞领域特定语言 Zooid，这是一个 Coq 库，用于验证消息传递代码是否遵循给定的多方会话类型（这类编排不含计算） [6]；以及多方 GV，一种目标与 Zooid 类似的形式化函数式语言 [17]。

6 结论

我们提出了首个针对编排的修正过程的形式化。我们的工作建立在对核心编舞及其配套的 EPP 概念进行的先前形式化之上，并将其作为库使用。我们发现，这一形式化具有模块化特性，且足够完整，能够支持本文所介绍的独立开发。秉持通用性与可复用性的相同理念，我们的形式化没有添加库中未包含的任何关于核心编舞的假设。

我们的开发展示了定理证明器如何协助开展研究：与 Coq 的交互引导我们（i）发现背景文献中为本工作提出的修正语义性质是错误的；以及（ii）找到一种正确的表述，并且该表述仍然足够强大，能够用于其在先前工作中的预期用途。

修订的形式化适合进行提取，因此我们的工作为构建一个经认证的转换器奠定了基础，该转换器可将核心编舞中的任意编排转换为可投影的编排。未来，我们计划研究如何将这一转换器集成到现有的编排式编程框架中。

我们的修订概念与 CC 的 EPP 定义有着内在联系。文献中存在对选择知识要求更宽松的编舞语言，例如，允许原本用于提供服务的复制进程只参与条件语句的一个分支 [2, 4]。研究如何将修订调整到这些情形中会很有意思。

参考文献

1 Saikat Bardhan Stekkelpul 和 Andrzej Wąsowski 编，*FASE* 论文集，第 9633 卷，计算机科学讲义，第 13–30 页。施普林格，2016年。 doi:10.1007/978-3-662-49665-7_2。2 Marco Carbone、Kohei Honda 和 Nobuko Yoshida。面向网络服务的结构化通信中心编程。*ACM* 程序设计语言与系统汇刊，34(2):8:1–8:78，2012。doi:10.1145/2220365.2220367。3 Marco Carbone、Sam Lindley、Fabrizio Montesi、Carsten Schürmann 和 Philip Wadler。相干性推广了对偶性：对多方会话类型的逻辑解释。收录于 Josée Desharnais 和 Radha Jagadeesan 编，*CONCUR* 论文集，第 59 卷，*LIPIcs*，第 33:1–33:15 页。达格施图尔城堡 - 莱布尼茨信息学中心，2016年。4 Marco Carbone 和 Fabrizio Montesi。设计上无死锁：多方异步全局编程。收录于 Roberto Giacobazzi 和 Radhia Cousot 编，*POPL* 论文集，第 263–274 页。ACM，2013年。 doi:10.1145/2429069.2429101。5 Giuseppe Castagna、Mariangiola Dezani-Ciancaglini 和 Luca Padovani。论全局类型与多方会话。收录于 Roberto Bruni 和 Jürgen Dingel 编，分布式系统形式化技术——第 *13* 届 *IFIP WG 6.1* 国际会议 *FMOODS 2011* 与第 *31* 届 *IFIP WG 6.1* 国际会议 *FORTE 2011* 联合会议，冰岛雷克雅未克，*2011*年*6*月*6*日至*9*日。会议论文集，第 6722 卷计算机科学讲义，第 1–28 页。施普林格，2011。 doi:10.1007/978-3-642-21461-5_1。

6 David Castro-Perez、Francisco Ferreira、Lorenzo Gheri 和 Nobuko Yoshida。Zooid：一种用于经认证的多方计算的领域特定语言：从机械化元理论到经认证的多方进程。收录于 Stephen N. Freund 和 Eran Yahav 编，PLDI 论文集，第 237–251 页。ACM，2021年。 doi:10.1145/3453483.3454041。7 Luís Cruz-Filipe 和 Fabrizio Montesi。编舞式编程的核心模型。收录于 Olga Kouchnarenko 和 Ramtin Khosravi 编，FACS 论文集，第 10231 卷《计算机科学讲义》，第 17–35 页。施普林格，2017年。 doi:10.1007/978-3-319-57666-4_3。8 Luís Cruz-Filipe 和 Fabrizio Montesi。编舞式编程的核心模型。《理论计算机科学》，802:38–66，2020年。 doi:10.1016/j.tcs.2019.07.005。9 Luís Cruz-Filipe、Fabrizio Montesi 和 Marco Peressotti。编舞编译认证。收录于 Antonio Cerone 和 Peter Csaba Ölveczky 编，ICTAC 论文集，第 12819 卷 LNCS，第 115–133 页。施普林格，2021年。 doi:10.1007/978-3-030-85315-0_8。10 Luís Cruz-Filipe、Fabrizio Montesi 和 Marco Peressotti。在 Coq 中形式化一种图灵完备的编舞语言。收录于 Liron Cohen 和 Cezary Kaliszyk 编，ITP 论文集，第 193 卷 LIPIcs，第 15:1–15:18 页。达格施图尔城堡—莱布尼茨信息学中心，2021年。 doi:10.4230/LIPIcs.ITP.2021.15。11 Luís Cruz-Filipe、Fabrizio Montesi 和 Marco Peressotti。编舞式编程的形式理论。CoRR，abs/2209.01886，2022年。网址： https://arxiv.org/abs/2209.01886。12 Mila Dalla Preda、Maurizio Gabbrielli、Saverio Giallorenzo、Ivan Lanese 和 Jacopo Mauro。动态编舞：理论与实现。《计算机科学逻辑方法》，13(2)，2017年。 doi:10.23638/LMCS-13(2:1)2017。13 Alain Finkel 和 Étienne Lozes。通信有限状态机的可同步性不可判定。收录于 Ioannis Chatzigiannakis、Piotr Indyk、Fabian Kuhn 和 Anca Muscholl 编，2017年7月10日至14日于波兰华沙举行的第 44 届自动机、语言与编程国际研讨会 ICALP 2017，第 80 卷 LIPIcs，第 122:1–122:14 页。达格施图尔城堡 - 莱布尼茨信息学中心，2017年。 doi:10.4230/LIPIcs.ICALP.2017.122。14 Saverio Giallorenzo、Fabrizio Montesi 和 Marco Peressotti。作为对象的编舞。CoRR，abs/2005.09520，2020年。网址： https://arxiv.org/abs/2005.09520。15 Andrew K. Hirsch 和 Deepak Garg。Pirouette：高阶类型化函数式编舞。《ACM 程序设计语言汇刊》，6(POPL):1–27，2022年。 doi:10.1145/3498684。16 Kohei Honda、Nobuko Yoshida 和 Marco Carbone。多方异步会话类型。《ACM 杂志》，63(1):9，2016年。另见：POPL，第 273–284 页，2008年。 doi:10.1145/2827695。17 Jules Jacobs、Stephanie Balzer 和 Robbert Krebbers。多方 GV：具有经认证的无死锁性的函数式多方会话类型。已获准发表于 ICFP 2022，2022年。18 Sung-Shik Jongmans 和 Petra van den Bos。用于编舞的谓词变换器——计算编舞式编程中的前置条件。收录于 Ilya Sergey 编，《编程语言与系统》——第 31 届欧洲编程研讨会，ESOP 2022，作为软件理论与实践欧洲联合会议 ETAPS 2022 的一部分举行，德国慕尼黑，2022年4月2日至7日。会议论文集，第 13240 卷《计算机科学讲义》，第 520–547 页。施普林格，2022 年。 doi:10.1007/978-3-030-99336-8_19。19 Stephen Cole Kleene。《元数学导论》，第 1 卷。北荷兰出版公司，1952年。20 Ivan Lanese、Fabrizio Montesi 和 Gianluigi Zavattaro。编舞修订。收录于 António Ravara 和 Josep Silva 编，2013年6月6日于意大利佛罗伦萨举行的第 9 届 Web 系统自动化规范与验证国际研讨会 WWV 2013 论文集，第 123 卷 EPTCS，第 34–48 页，2013年。 doi:10.4204/EPTCS.123.5。21 Tanakorn Leesatapornwongsa、Jeffrey F. Lukman、Shan Lu 和 Haryadi S. Gunawi。Taxdc：数据中心分布式系统中非确定性并发错误的分类体系。收录于 Tom Conte 和 Yuanyuan Zhou 编，2016年4月2日至6日于美国佐治亚州亚特兰大举行的第二十一届编程语言与操作系统架构支持国际会议 ASPLOS 2016 论文集，第 517–530 页。ACM，2016年。 doi:10.1145/2872362.2872374。

22 Alberto Lluch-Lafuente、Flemming Nielson 和 Hanne Riis Nielson。面向交互规约的自由裁量信息流控制。载于 Narciso Martí-Oliet、Peter Csaba Ölveczky 和 Carolyn L. Talcott 编，《逻辑、重写与并发》，计算机科学讲义第 9200 卷，第 427–450 页。施普林格，2015年。doi:10.1007/978-3-319-23165-5_20。23 Hugo A. López 和 Kai Heussen。为能源领域编排信息物理分布式控制系统。载于 Ahmed Seffah、Birgit Penzenstadler、Carina Alves 和 Xin Peng 编，SAC 论文集，第 437–443 页。ACM，2017年。 doi:10.1145/3019612.3019656。24 Fabrizio Montesi。《编排式编程》。博士论文，哥本哈根信息技术大学，2013。网址：http://www.fabriziomontesi.com/files/choreographic_programming.pdf。25 Fabrizio Montesi。《编舞简介》。剑桥大学出版社，2023年。26 Roger M. Needham 和 Michael D. Schroeder。使用加密在大型计算机网络中进行身份验证。《ACM 通讯》，21(12)：993–999，1978年。 doi:10.1145/359657.359659。27 Johannes Åman Pohjola、Alejandro Gómez-Londoño、James Shaker 和 Michael Norrish。Kalas：一种经过验证的端到端编舞式语言编译器。载于 June Andronick 和 Leonardo de Moura 编，《第 13 届交互式定理证明国际会议（ITP 2022）》，2022年8月7日至10日，以色列海法，第 237 卷 LIPIcs，第 27:1–27:18 页。Schloss Dagstuhl - Leibniz-Zentrum für Informatik，2022年。doi:10.4230/LIPIcs.ITP.2022.27。28 艾伦·图灵。可计算性与 λ-可定义性。《符号逻辑杂志》，2(4)：153–163，1937年。doi:10.2307/2268280。